Tolerating Process Variations in Large, Set-Associative Caches: The Buddy Cache

CHENG-KOK KOH

Purdue University

WENG-FAI WONG

National University of Singapore

and

YIRAN CHEN and HAI LI

Seagate Technology LLC

One important trend in today's microprocessor architectures is the increase in size of the processor caches. These caches also tend to be set associative. As technology scales, process variations are expected to increase the fault rates of the SRAM cells that compose such caches. As an important component of the processor, the parametric yield of SRAM cells is crucial to the overall performance and yield of the microchip. In this article, we propose a microarchitectural solution, called the buddy cache that permits large, set-associative caches to tolerate faults in SRAM cells due to process variations. In essence, instead of disabling a faulty cache block in a set (as is the current practice), it is paired with another faulty cache block in the same set—the buddy. Although both cache blocks are faulty, if the faults of the two blocks do not overlap, then instead of losing two blocks, buddying will yield a functional block from the nonfaulty portions of the two blocks. We found that with buddying, caches can better mitigate the negative impacts of process variations on performance and yield, gracefully downgrading performance as opposed to catastrophic failure. We will describe the details of the buddy cache and give insights as to why it is both more performance and yield resilient to faults.

Categories and Subject Descriptors: B.3.4 [Hardware]: Memory Structures—*reliability, testing and fault-tolerance*

General Terms: Processor caches, process variation, reliability

Additional Key Words and Phrases: Processor architectures, caches, memory structures, fault recovery

ACM Reference Format:

Koh, C.-K., Wong, W.-F., Chen, Y., and Li, H. 2009. Tolerating process variations in large, set-associative caches: The buddy cache. *ACM Trans. Architect. Code Optim.* 6, 2, Article 8 (June 2009), 34 pages. DOI = 10.1145/1543753.1543757 <http://doi.acm.org/10.1145/1543753.1543757>.

Author's addresses: C-K Koh, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907; email: chengkok@ecn.purdue.edu; W-F Wong, School of Computing, National University of Singapore; email: wongwf@comp.nus.edu.sg.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1544-3566/2009/06-ART8 \$10.00
DOI 10.1145/1543753.1543757 <http://doi.acm.org/10.1145/1543753.1543757>

1. INTRODUCTION

One of the main benefits that deep submicron or nano-scale technology offers is higher density. Much of this density is deployed in the processor, bringing about the advent of multicore technology, and/or larger caches. Current generation of micro-processors already have megabytes of mainly set-associative caches on-chip, often occupying half of the silicon real estate. With multicore processors, locality in caches becomes an important method for alleviating off-core and off-chip traffic. We therefore expect large caches to remain a feature.

The flip side of the coin, however, is that at 45nm technology and beyond, process parameter variations are expected to severely affect the performance and yield of such large-scale caches [Lee et al. 2007; Ozdemir et al. 2006; Agarwal et al. 2005; Fischer et al. 2007]. Operating large caches at low voltages, as is often required for energy conservation, also introduces error [Wilkerson et al. 2008]. At the micro-architectural level, such faults are starting to drive research in fault tolerant designs. For memory design, redundant rows/columns are often used to replace faulty rows/columns. Error Correcting Code (ECC) can also be used to correct errors due to faulty memory cells. However, ECC has significant area overhead and computation complexity if correction of multiple errors is required [Wilkerson et al. 2008]. Hence, most existing ECC schemes correct only a single error. Agarwal et al. [2005] showed that simple redundancy techniques and ECC schemes are quite ineffective in coping with a large number of faulty cells in a large modern cache. Moreover, using ECC to correct failures due to process variations divests a cache of its ability to correct transient faults (or soft errors), the primary function performed by ECC in modern-day memory subsystems. In this article, we shall consider the challenge of making large, set-associative cache more resilient to faults arising from defects in silicon, lithography-based failures, and parametric variations. We do so by the introduction of the buddy cache, a novel microarchitecture design that recycles and reuses faulty cache blocks.

Earlier works on fault-tolerant cache design generally took advantage of the inherent ability of especially set-associative caches to tolerate faults. In previous works [McClure 1997 and Arimilli et al. 1999], a cache block is turned off when they are found to be faulty. Other cache blocks in different ways of the same set of the disabled cache block are still functional. We call techniques such as these together as downsizing by discarding faulty blocks (DDFB). We note that such schemes require extra circuitry to disable faulty blocks and redirect the access to a disabled cache block to a functional one. In the example of a direct-mapped L1 cache used in Agarwal et al. [2005], individual faulty cache blocks can be turned off by means of a bit stored in a fault map, and the access to a disabled cache block is redirected by column multiplexing to a functional one in the same row. In PADded cache [Shirvani and McCluskey 1999], a programmable address decoder remaps references to a faulty block to a functional one in a different set, but in the same way of memory array. The programmable address decoder also requires the equivalence of a fault map for proper configuration. We, therefore, consider it to be similar to our canonical DDFB scheme. A fairly drastic approach is taken by Ozdemir et al. [2006]: if a cache way

violates the maximum allowable latency or a power constraint, the cache way as well as its corresponding decoders, precharge circuits, and sense-amplifier are all turned off. They called it “yield-aware power-down” (YAPD). This design eliminates the fault map by sacrificing an entire way of the cache. A commercial instance of DDFB is Intel’s Pellston technology [Pellston 2006] which was first implemented in the L3 cache of the Itanium processor. Using ECC, a double-bit error in a 128-byte cache block is detected and the block is disabled.

DDFB applied within a set is, however, quite wasteful. Consider a block size of 64 bytes, that is, 512 bits, which is a common block size in today’s caches. For bit fault probabilities that are typically in the order of 10^{-4} to 10^{-3} [Agarwal et al. 2005], the likelihood of more than 1 bit at fault is relatively low. We would prefer to have a way of recovering most of the 511 nonfaulty bits for use. The key insight in our proposed buddy cache design is that in addition to this fact, it is also rare that two faulty blocks in a set have faults in exactly the same bit position. Hence, rather than discarding both blocks, we can “buddy” them to act like a single nonfaulty block. Based on this idea, this article will describe the microarchitectural implementation of the buddy cache and assess the impact of buddying on performance and yield:

- (1) Instead of discarding faulty blocks, the buddy cache reuses faulty blocks to achieve a higher number of functional blocks within a cache set. Buddy cache design therefore improves the performance of large, set-associative caches over DDFB, significantly.
- (2) The remapping logic of the buddy cache for the identification of buddy blocks is robust and inherently fault tolerant. Buddy cache design can, therefore, maintain high yield even under high-fault probabilities.

Two schemes proposed recently that are comparable to our proposal are the word-disable (WDIS) and the bit-fix (BFIX) schemes [Wilkerson et al. 2008]. The WDIS scheme combines two consecutive cache blocks into a single cache block, thereby reducing the capacity by 50%, whereas the BFIX scheme sacrifices a cache block to repair defects in three other cache blocks, thereby reducing the capacity by 25%. Both schemes attempt to fix faults and do not reuse faulty blocks in the way that the buddy cache does. As a result, functionally perfect blocks may be sacrificed unnecessarily. We shall show that our buddy cache implementation enjoys significant advantages over both WDIS and BFIX schemes in both performance and yield of large set-associative caches. Moreover, the performance and yield of the buddy cache degrade gracefully in the presence of faults, whereas the degradations experienced by the WDIS and BFIX schemes are quite drastic.¹

The rest of the article is organized as follows: We begin with a survey of related works in Section 2. The microarchitectural design of the buddy cache will be described in detail in Section 3. From the fault configurations generated from Monte Carlo simulations, we show evidence in Section 4 that buddying

¹*Caveat lector:* The work in [Wilkerson et al. 2008] was aimed at reducing the minimum voltage at which large set-associative caches can reliably operate so as to reduce power consumption.

makes the overall system more resilient to performance degradation due to faults in the caches. In Section 5, we show with analytical yield models as well as empirical data obtained from Monte Carlo simulations that buddy cache has inherently high yield. We will then explore some of the design parameter issues of the buddy cache in Section 6. This is followed by a conclusion.

2. RELATED WORKS

Due to the importance of the issues, there is a large body of literature dealing with process variations and yield, especially at the circuit level [Borkar et al. 2003; Borkar 2005; Tschanz et al. 2005; Choi et al. 2004]. Techniques for tolerating process variations in the processor pipeline and architectural level have also been actively researched and proposed [Datta et al. 2005; Liang and Brooks 2006; Xiong et al. 2005], including the use of globally synchronous, locally asynchronous designs [Marculescu and Talpes 2005]. The yield of SRAM storage cells has also been the subject of several studies [Mukhopadhyay et al. 2004; Kurdahi et al. 2006]. However, these techniques are at what are considered to be a lower level than (and orthogonal to) what we will propose.

The turning off of sets or ways in set-associative caches for purposes such as energy saving has attracted a fair amount of attention [Powell et al. 2001; Zhang et al. 2005]. We found two patents [McClure 1997; Arimilli et al. 1999] that proposed circuits to bypass faulty cache blocks in a set-associative cache. Another patent filed in 1998 by Fujimoto [2000] achieves the same effect by using the pseudo-LRU bits. We consider these to be the original DDFB schemes in addition to the studies by Shirvani and McCluskey [1999], Agarwal et al. [2005], and Ozdemir et al. [2006] we have mentioned in Section 1. More recently, Lee et al. [2007] described a faulty cache simulation tool, CAFÉ as well as a variety of DDFB strategies to deal with faults. They also made a good case for the need to “gracefully degrade” the cache in the presence of faults. None of these schemes, however, considered the recycling of faulty blocks to boost both yield and performance, the basic idea behind this work.

The WDIS scheme and BFIX scheme proposed in a recent study [Wilkerson et al. 2008] come close to being comparable with our proposal. In the WDIS scheme, two consecutive cache blocks are combined into a single cache block. A 64-byte cache block, for example, is divided into 16 32-bit words. With 1 bit to record whether a 32-bit word is functional, a 16-bit fault map is stored alongside the tag. In a read operation, two consecutive cache blocks are accessed simultaneously. Then, a four-stage shifter removes the defective word from a cache block based on the fault map to reconstruct one half of a full cache block. Consequently, the capacity of the cache is reduced by 50%.

Whereas the WDIS scheme performs the necessary repairs at the (32-bit) word level and within a cache block, the BFIX scheme performs the necessary patchings for pairs of bits and stores the patches of three cache blocks in a cache block residing in another bank. A cache block that stores the patches is called a *fix line*, and it stores 10 patches for a cache block that it is paired with. Each patch contains the address of the faulty 2-bit group and the correct repair patterns. To protect against defects in a patch, a single-bit error correction code

is used. In a read operation, the cache block that corresponds to a tag match and its fix line are accessed simultaneously. The patches are decoded and a 10-stage shifter removes defective 2-bit groups out of the tag-matched cache block. As a cache block can be used as a fix line for three other cache blocks, the BFIX scheme reduces the cache capacity by 25%.

In [Wilkerson et al. 2008], the defects are introduced because of low-voltage operation. As the supply voltage is scaled, more SRAM cells become faulty. Both WDIS and BFIX schemes allow a large set-associative cache to remain functional, albeit with a reduced capacity, with even lower supply voltage. Therefore, the power consumed by a micro-processor can be reduced. There are a few fundamental differences that between these schemes and the buddy cache that would become more apparent in the remainder of the article:

- In [Wilkerson et al. 2008], the tag array in both WDIS and BFIX and the fault map in WDIS are assumed to be perfect, because a different SRAM circuitry that has a larger silicon footprint is used in their implementations. The buddy cache operates even with faulty tag array and fault map.
- In a buddy cache, a functional cache block remains standalone and does not have to be paired with an adjacent cache block or a fix line as in the case of the WDIS and BFIX schemes, respectively.
- Both WDIS and BFIX require higher-access cycle latency because they perform the correction of a cache block *after* data access. The overhead of the buddy cache is incurred mainly during the tag match phase, thereby allowing some of the overhead to be hidden.

3. THE BUDDY CACHE

Consider an n -way set-associative cache with m sets. Let the n blocks within a set be labeled $B_0, B_1, \dots, B_{n-1}$. We divide each cache block, say B_i , into k divisions, with each division guarded by a fault bit in a fault map. In other words, if a division is faulty, the corresponding fault bit for that division is set to “1”. The fault bit of an operational division is set to “0”. The tag field of a block is guarded by a separate bit. Therefore, there are $k + 1$ fault bits for a block. In an n -way set-associative cache with m sets, there are a total of $(k + 1)nm$ bits in a fault map.

Let $f_{i0} f_{i1} \dots f_{i(k-1)} f_{ik}$ denote the $k + 1$ fault bits of cache block B_i and its tag. Suppose some of the k divisions in a block and/or its tag are faulty. Consider another block B_j within the same set with $f_{j0} f_{j1} \dots f_{j(k-1)} f_{jk}$ denoting its $k + 1$ fault bits. If the bit-wise “AND” operation of the bit sequences $f_{i0} f_{i1} \dots f_{i(k-1)} f_{ik}$ and $f_{j0} f_{j1} \dots f_{j(k-1)} f_{jk}$ results in a sequence of all 0 bits, the pairing of the two blocks will produce a faultless combination that can be used to store both data and tag. The selection of a buddy block is done during initialization and will be discussed in Section 3.3.

The proper operation of a buddy cache requires a buddy map to store the index of the buddy block of a faulty block. Given that there are n blocks within a set, $\log n$ bits are required to encode the index of a block. Therefore, $n \log n$ bits are required to store the pairing information within a set. For simplicity,

Table I. An Example of the Fault and Buddy Map Entries

Block	Divisions	Tag	Fault Map	Buddy Map
0	$D_{00}D_{01}D_{02}D_{03}$	T_0	0 0 1 0 0	1 0
1	$D_{10}D_{11}D_{12}D_{13}$	T_1	0 0 0 0 0	0 1
2	$D_{20}D_{21}D_{22}D_{23}$	T_2	1 1 0 1 1	0 0
3	$D_{30}D_{31}D_{32}D_{33}$	T_3	0 0 0 0 0	1 1

we assume that each set of the cache has a corresponding row in the fault map and buddy map. We shall also assume that these two maps are placed adjacent to one another in a combined fault and buddy map. For a fully operational cache block, its buddy map entry simply points to itself. For a faulty block, if a buddy block can be found within the set, the index of the buddy block is stored in the buddy map. If a buddy block cannot be found, we assign 1's to all the fault bits for that block in the fault map. In addition, we store the index of the block in its own buddy map entry, thereby disabling the faulty block.

We shall use a 4-way buddy cache design to illustrate the idea. Without loss of generality, assume that the width of every data block is 64 bytes and that each data block is divided into 4 divisions, that is, each division has 16 bytes. Let the four divisions of block i be labeled D_{i0} , D_{i1} , D_{i2} , and D_{i3} . Moreover, let T_i denote the tag of block B_i . Let us buddy up B_0 and B_2 , say, assuming that divisions D_{02} , D_{20} , D_{21} , and D_{23} and tag T_2 have faults in them. The combined fault and buddy map of the blocks are shown in Table I.

We assume that the fault maps of two buddy cache blocks, say B_i and B_j , are complementary (i.e., $f_{iq} \oplus f_{jq} = 1$ for $0 \leq q \leq k$). For example, consider the buddy pair B_0 and B_2 shown in Table I. Suppose that only divisions D_{02} and D_{23} are faulty. Even though divisions D_{00} and D_{20} as well as D_{01} and D_{21} , and tags T_0 and T_2 are faultless, we have to set one of the fault bits from each pair of divisions and tags to “1.” For example, valid fault map entries for B_0 and B_2 would be “00100” and “11011,” respectively.

3.1 Implementation of Buddy Cache

Figure 1 shows a conventional 4-way cache design. The signals from the decoder select the corresponding tag value and data block from four different ways in the same set. The result of comparing the stored tags with the incoming address tag together with the validity bit generate the selection signal Sel_i , $i \in \{0, 1, 2, 3\}$. All four Sel_i signals are sent into a 4-to-1 MUX to select the final output from the four data blocks. Also, if any Sel_i signal is high, a “hit” signal is asserted.

To implement a k -division buddy cache, we will make the following changes to an n -way, l bytes per block set-associative cache (see Figure 2—parts of the diagram has been omitted for clarity). In place of the n -to-1 l -byte MUX, k n -to-1 (l/k) -byte MUXs are needed. Therefore, for every (l/k) -byte division, we have n selection inputs sel_{ij} to select D_{ij} , the j th division of block (way) B_i , $i \in \{0, \dots, n-1\}$, $j \in \{0, \dots, k-1\}$. Let Sel_i , $i \in \{0, \dots, n-1\}$, be asserted when the incoming tag matches a valid stored tag T_i . To ensure that the matched tag is from an operational tag, we define $\hat{Sel}_i$ as $(Sel_i \cdot \overline{f_{ik}})$, where f_{ik} is the fault bit of the tag. If any $\hat{Sel}_i$ signal in a set is high, a “hit” signal is asserted.

Fig. 1. A conventional 4-way set-associative cache.

The assertion of $\hat{Sel}_i$ would be sufficient to select any, if not all, of k divisions within the block. However, if it has a buddy block, we also have to select some divisions within the buddy block. We rely on the buddy index of B_i , denoted by b_i . Let the n decoded buddy index bits be e_{ij} , $j = 0, \dots, n-1$. If $b_i = p$, that is, B_p is the buddy of B_i , $e_{ip} = 1$ while $e_{iq} = 0$ for all $q \neq p$, that is, at most one e_{ij} can be true. Let $bSel_{ij} = \hat{Sel}_i \cdot e_{ij}$. Each of the $bSel_{ij}$ signal is being routed from way i to way j , except for $bSel_{ii}$, which is discarded, since we have $\hat{Sel}_i$.

At each way i , we have the following signals:

$$bSel_{0i}, \dots, bSel_{(i-1)i}, \hat{Sel}_i, bSel_{(i+1)i}, \dots, bSel_{(n-1)i}$$

The assertion of any of these signals implies that we should select some (operational) divisions from the block B_i . Therefore, we define sel_{ij} , the signal for the selection of D_{ij} , for $i \in \{0, \dots, n-1\}$ and $j \in \{0, \dots, k-1\}$, as

$$\begin{aligned} sel_{ij} = & (bSel_{0i} + \dots \\ & + bSel_{(i-1)i} + \hat{Sel}_i + bSel_{(i+1)i} \\ & + \dots + bSel_{(n-1)i}) \cdot \overline{f_{ij}}. \end{aligned} \quad (1)$$

Fig. 2. Way 0 of a 4-division, 4-way set-associative, 64 bytes per block buddy cache.

When the cache is faultless, we do not want to incur the energy or performance overhead of the budding mechanism. We achieve that by using a single bit Z as an enabling signal for the combined fault and buddy map. This Z bit is set during initialization (see Section 3.3) when the cache is found to be faultless. In this case, the combined fault and buddy map are not enabled and the outputs from it are pulled to ground (i.e., zero). This ensures that should the cache be faultless, no dynamic energy or performance overhead is incurred by the budding mechanism.

In our implementation of the buddy cache, we assume that like a conventional cache, the buddy cache is protected from transient faults by ECC. In other words, the combined fault and buddy map handles faults in SRAM cells due to defects in silicon, process variations, and the like. ECC deals only with soft errors.

3.2 Remapping Logic: Handling Faults in the Combined Fault and Buddy Map

The preceding discussion has assumed a perfect combined fault and buddy map. However, the combined fault and buddy map suffers the same fault rates as the cache. We shall now describe a remapping scheme to ensure that even when the combined fault and buddy map is faulty, the buddy cache is still operational.

Consider the fault bits of a block in the fault map and the buddy index of the same block in the buddy map. If the fault bits and the buddy index are both

Table II. Fault Combinations for the Combined Fault and Buddy Map, and the Cache Blocks

Fault map of block		Buddy map of block	Cache block	Outcome
Tag	Data (k bits)			
Perfect	Perfect	Perfect	Don't care	Cache block usable.
Perfect	Perfect	Faulty	Perfect	Cache block perfect.
			Faulty	All bits in fault map entry assigned 1. Cache block not usable.
Perfect	>1 bit faulty	Don't care	Don't care	Nonfaulty bits in fault map entry assigned 1. Cache block not usable.
Faulty	< k bits faulty	Perfect	Don't care	Nonfaulty bits in fault map entry assigned 1. Buddy address points to self. Cache block not usable.
Faulty	Don't care	Faulty	Don't care	Row not usable
Faulty	All bits faulty	Don't care	Don't care	Row not usable

operational, the status of the corresponding block can be properly recorded using the fault bits and the buddy index.

If the fault bits are operational but the buddy index is faulty, a fully operational cache block can still be recorded properly in the fault bits (regardless of the status of the faulty buddy index). However, if the tag or any of the data divisions is faulty, we assign “1” to all the corresponding fault bits of the block in the fault map. In other words, a faulty cache block that also has a faulty buddy map entry cannot be buddied with another block and is disabled. When the fault bit for the tag is functional and some fault bits for the data divisions are faulty, we disable the block by setting the nonfaulty fault bit for the tag to “1,” while making sure that the block is not buddied with another faulty block. In the case where the fault bits are faulty but the buddy index is not faulty, we also have a way of indicating that this is a data cache block that should not be used as long as at least one of the fault bits is not faulty. We assign the nonfaulty bit(s) to “1” and store the index of the faulty block in the buddy index.

However, when all the fault bits of a cache block are faulty, we have to discard the entire row of the combined fault and buddy map and use a spare row. Moreover, we also have to use a spare row when both the fault bit for a tag and the buddy address bits are faulty. Table II summarizes the scenarios we have just described.

In order to tolerate faulty cells in the combined fault and buddy map, we must be able to recover the correct fault bits and buddy index from faulty ones. In particular, we have to consider the following scenarios: (i) the fault map is perfect but the buddy map is faulty (second entry of Table II), (ii) the fault bit for the tag is perfect, but some fault bits for the data divisions are faulty (third entry of Table II), and (iii) the buddy map is perfect and there are $<k+1$ faulty fault bits (fourth entry of Table II). The remaining scenarios in Table II are trivial; either the combined map is perfect or the corresponding row in the combined map would have been discarded during the configuration phase.

Now, let f_{ij} and b_i be the correct fault bits and buddy index, respectively. Let $\tilde{f}_{ij}$ and $\tilde{b}_i$ denote the stored fault bits and buddy index, which may be incorrect. To generate b_i , we use a 2-to-1 MUX (that is $\log n$ bits wide), whose inputs are i and $\tilde{b}_i$. We shall assume that the way number i is a hard-wired constant. The selection signal to the MUX is the “OR” of $\tilde{f}_{ij}$. In other words,

$$b_i = (\tilde{f}_{i0} + \dots + \tilde{f}_{ik}) \cdot i + (\tilde{f}_{i0} + \dots + \tilde{f}_{ik}) \cdot \tilde{b}_i. \quad (2)$$

An equivalent but more efficient approach to generate the correct b_i is to use $(\tilde{f}_{i0} + \dots + \tilde{f}_{ik})$ as the enabling signal for the decoder for b_i .

To generate f_{ij} , we check whether the buddy index $\tilde{b}_i$ is i . The correct fault bit f_{ij} can be obtained as follows:

$$f_{ij} = (\tilde{b}_i == i) \cdot (\tilde{f}_{i0} + \dots + \tilde{f}_{ik}) + \tilde{f}_{ij}, \quad (3)$$

where the operator “==” indicates equality check, which returns “1” when both of its operands are equal.

We now examine whether we have recovered the correct fault bits and buddy indices. Also, it is important that we do not introduce errors into fault bits and buddy indices that are correct.

- Fault map perfect, buddy map perfect.* If the cache block B_i is perfect, all of its $\tilde{f}_{ij} = 0$ and $\tilde{b}_i = i$. Then $f_{ij} = \tilde{f}_{ij}$ and $b_i = \tilde{b}_i$. If B_i is faulty, some of its $\tilde{f}_{ij} = 1$ but $\tilde{b}_i \neq i$. Again, $f_{ij} = \tilde{f}_{ij}$ and $b_i = \tilde{b}_i$.
- Fault map is perfect, buddy map is faulty.* If B_i is perfect, $f_{ij} = 0$, regardless of $\tilde{b}_i$. More important, $b_i = i$ by Eq. (2) ensuring the correct decoding of e_{ij} . If B_i is faulty, all $\tilde{f}_{ij} = 1$. Regardless of the value of $\tilde{b}_i$, $f_{ij} = 1$ by Eq. (3). Consequently, none of the divisions in the block are accessible. Moreover, even if the buddy index does not point to itself, data divisions in other blocks will not be incorrectly selected as $\hat{\text{Sel}}_i$ is negated because f_{ik} , the tag’s fault bit, is 1.
- Tag fault bit perfect, some fault bits for data divisions are faulty.* By Eq. (3), the tag fault bit $f_{ik} = 1$. Consequently, $\hat{\text{Sel}}_i$ is negated, which ensures that B_i would not be buddied with another block since $\text{bSel}_{ij} = 0$, for $j \in \{0, \dots, n-1\}$ and $i \neq j$. As long as no other blocks use block i as a buddy, $\text{bSel}_{ji} = 0$, for $j \in \{0, \dots, n-1\}$. Consequently, $\text{sel}_{ij} = 0$ for $j \in \{0, \dots, k-1\}$. In other words, B_i is disabled.
- Tag fault bit faulty, $< k$ other fault bits faulty, buddy map is perfect.* Since at least one $\tilde{f}_{ij} = 1$, all $f_{ij} = 1$ by Eq. (3), thereby disabling B_i .

Figure 3 is a modified version of Figure 2 in which the remapping logic described to handle the possibility of faults appearing in the combined fault and buddy map has been incorporated.

Writing to a buddy cache is similar to reading it. During a write, the appropriate bit lines are selected based on a similar logic in which bit lines are selected during a read operation. In other words, although we have to write to two buddied faulty blocks, we use the same number of bit lines as in a conventional cache. Hence, writes to buddied faulty blocks suffer neither performance nor energy penalties.

Fig. 3. Buddy cache with remapping logic to handle faults in combined fault and buddy map.

Fig. 4. Modified pseudo-LRU of the buddy cache.

For replacement, the normal operations of pseudo-LRU algorithm is not affected by buddying—if a buddied block is chosen for replacement, the read and write circuitry will ensure that the correct data block is assembled from the selected block and its buddy. What is needed is to ensure that a discarded block is not chosen. We adopt a technique proposed by Fujimoto [2000]. Figure 4 shows a modified tree-LRU circuit for a 4-way set-associative cache. Each LRU bit determines if the search for the LRU block should go left (if the bit is 0) or right

(if it is 1). After an access, the tree is traversed from the LRU leaf node back to the root, complementing the LRU bits along the way. This will effectively send the next search for the LRU block to the “other” subtree. In our example, suppose (only) Block 1 is faulty, then f_{14} , the fault bit for the tag of that block computed by the remapping logic, is always set. This in turn forces LRU Bit 1 to be always set to 0, i.e. the left block, Block 0. Therefore, Block 1 will never be accessed. If both Block 0 and 1 are faulty, then f_{04} and f_{14} together will force LRU Bit 0 to always direct the search to the right subtree at the root. Note that we show only the first 3 input signals to each MUX in Figure 4. When both selection signals are 1, we do not select any blocks below the corresponding LRU bit, i.e., the 4th input to the MUX is a “Don’t Care”. Instead, the LRU bit at one level higher of the hierarchy will direct the search to occur in the other branch. In our implementation, we require each set to have at least one functional block (standalone or buddied). Therefore, the 4th input to any MUX in the tree-LRU circuit will never be selected.

To identify the buddied blocks and the corresponding divisions of a pair of buddied blocks, we use a logic very similar to Figure 3. Again, the main challenge here is the generation of the bit $bSel_{ij}$. For the computation of that signal bit, all we have to do is redefine $\hat{Sel}_i$. In Figure 3, for example, $\hat{Sel}_0$ for the read operation is the “AND” of the valid flag, the tag match signal, and $\overline{f_{04}}$. For the write operation, first we assume that when the LRU circuitry selects block 0, it also asserts the signal LRU_0 . Now, we redefine $\hat{Sel}_0$ as $LRU_0 \cdot \overline{f_{04}}$ for a write operation. In general, we redefine $\hat{Sel}_i$ as $LRU_i \cdot \overline{f_{ik}}$, where LRU_i is the signal asserted when block i is selected by the LRU circuitry.

3.3 Configuring the Combined Fault and Buddy Map

Now, the remaining question is how the combined fault and buddy map can be configured properly. The testing of SRAM cells in the cache (tag and data) and the combined fault and buddy map is carried out during processor initialization using the traditional built-in self-test (BIST) approach for memory, as proposed by Agarwal et al. [2005]. A straightforward implementation of the configuration logic would involve registers to store the results of the built-in self-test of each set of cache lines and the corresponding bits in the combined fault and buddy map. Besides correcting permanent, process variation related faults, it also allows buddying to adapt to faults that may be introduced as the processor ages.

If all memory cells are functional, Z is set and the combined fault and buddy map are disabled. If there are faulty memory cells, the configuration of a row in the combined fault and buddy map is performed in the following four phases:

- (1) We first check whether the row is nonoperational. A row is nonoperational (regardless of data/tag) if (i) the $(k+1)$ fault bits of a cache block are all faulty or (ii) there are faults in both the tag fault bit and buddy index of a cache block. A redundant row in the combined fault and buddy map is used in that case. The combined fault and buddy map and hence, the buddy cache and the chip are rendered nonoperational if we have exhausted all redundant rows. Only simple “AND” and “OR” gates are required in this step (and the

Table III. Cache Access Delay Analysis

	Tag access delay (ps)	Data access delay (ps)	Total access time (ps)
32KB L1 conventional	277.7	415.3	687.8
4MB L2 conventional	368.7	1,902.7	3,188.6
32KB L1 buddy	388.8	415.3	687.8
4MB L2 buddy	519.9	1,902.7	3,339.8

next two steps). We use, for example, an “AND” gate to determine whether all $(k + 1)$ bits of a set in the combined fault and buddy map are faulty.

- (2) After the previous phase, we have eliminated the last two scenarios in Table II. In the second phase, we remove cache blocks that cannot be used because (i) some but not all of its fault bits are faulty and its buddy index is fully functional, or (ii) the tag fault bit and buddy index are faulty. We assign “1” to all the fault bits and point the buddy index back to the cache block itself (although only the operational fault bits and buddy index would be set properly).
- (3) Now, the remaining yet-to-be-assigned fault bits in the row are fully operational. We set them according to the functionality of the corresponding data divisions and tags in the cache. For those perfect cache blocks (data and tag), we set the buddy index to point to itself. For those imperfect cache blocks with faults in their buddy indices, we disable them by assigning “1” to all the fault bits.
- (4) In the final phase, we are left with imperfect cache blocks that have operational buddy indices. Proceeding from the lowest indexed cache block to the highest, we use a priority encoder to find for each imperfect cache block, a compatible imperfect cache block. To find the lowest indexed compatible cache block for an imperfect cache block i , we can use “OR” gates to determine whether there is a functional division or tag between cache block i and cache block j , where $j > i$. We can then use an “AND” gate to determine whether blocks i and j can be combined to operate as a functional block. The output of this “AND” gate is supplied to a priority encoder, which also takes in the output signals of “AND” gates that correspond to other higher indexed imperfect cache blocks $i < k \neq j$. The priority encoder will therefore select the lowest indexed cache block that is compatible to i . For every pair of buddies, we set the fault bits such that they are complementary.

If there are no operational cache blocks at the end of the process, a replacement row is used and steps 3 and 4 are repeated. If all redundant rows are exhausted, the cache, and hence the chip, is deemed nonoperational.

3.4 Timing Analysis of Buddy Cache

From Figure 3, it is clear that the tag access path is lengthened in a buddy cache design—the generation of the sel_{ij} signals relies on Sel_i , a signal that both the conventional and buddy caches generate. However, as we can see from the tag and data access delays of conventional caches shown in Table III, the generation of the Sel_i signal in both conventional and buddy caches occurs much earlier than the arrival of data at the respective multiplexers. The cache

access times for 32KB 8-way L1 and 4MB 16-way L2 caches were obtained based on implementations of various components of a cache, including the signal driver, cache line, sense amplifier, decoder, and other necessary logic gate, using the PTM 45nm technology [Cao et al. 2000]. The timing parameters for these components were obtained using SPICE simulations at a supply voltage of 0.9V. These timing parameters are in turn used in simulations of conventional L1 and L2 caches with CACTI version 4.2 [Tarjan et al. 2006] to obtain the respective cache access delays of Table III. Here, we assume that in L1 cache, both tag and data arrays are accessed simultaneously so as to reduce the access latency. In L2 cache, the tag array is accessed first in order to minimize energy. When there is a tag match, the corresponding block in the data array is activated to complete the read.

In the case of L1 cache, owing to the difference between the arrival times of Sel_i and the data, it is possible to hide the latency of the longer tag access path in a buddy cache with the available timing margin if the sel_{ij} signals can arrive at the multiplexers before the data D_{ij} do. It is evident from Table III that in a buddy cache, the critical portion of the tag access path, namely the generation and the propagation of bSel_{ij} to data division D_{ij} from tag T_i , can be completed before the arrival of D_{ij} . We kept the aspect ratio of the combined fault and buddy map to be close to 1 so that the access of the fault bits and buddy indices is not on the timing-critical path. The additional delay overheads in tag access for 32KB 8-way buddy cache with 4 data divisions is 111.1ps. Consequently, although the buddy cache design increases the tag access delay, the additional delay is hidden by the long access time of data, resulting in identical cache access delays for both the conventional and buddy caches. The same is true of the “fast path” when the buddy cache is faultless and the Z signal is asserted.

In the case of L2 cache, due to the sequential access of tag and data, it is not possible to hide the latency of the longer tag access path in a buddy cache. Consequently, the total access time of a 4MB 16-way buddy cache with 4 data divisions shown in Table III includes an additional delay in tag access of 151.2ps. That translate into one additional clock cycle for processors operating at up to 6.6GHz.

The remapping logic proposed in Section 3.2 for the identification of appropriate data divisions that are buddies takes advantage of the relatively long data access delay of L1 cache to avoid delay penalty. As data access delay and tag access delay are design-dependent, it is possible that tag access is on the timing-critical path instead. In that case, a different remapping logic is needed to hide the delay. If both data access and tag access are on the critical path, however, we would not have any slack to hide the remapping delay, and we may have to incur an additional cycle even for L1 cache access. However, this is not the case in the CACTI cache architecture that we used.

3.5 Area Overhead of Buddy Cache

For an n -way buddy cache with m sets of cache blocks, let t denote the size of a tag identifying a cache block. We shall assume that the data of a cache block is divided into k divisions, with each division of data guarded by a fault bit in the fault map. Let d denote the number of bits in a data division. Therefore, a cache

block would have $d \times k$ data bits. A buddy cache (and a conventional cache) has $mn(t + kd)$ bits in total in the tag and data arrays. In contrast, the combined fault and buddy map has a small area overhead of $mn(k + 1 + \log n)$ bits. The areas for a 4-data-division 32KB L1 buddy cache and a 4-data-division 4MB L2 buddy cache, normalized to that of a conventional cache, are 1.015 and 1.017, respectively. To put these numbers in perspective, the size of the combined fault and buddy for a 4-data-division 32KB L1 and 4-data-division 4MB L2 are 40% and 64% of the respective tag arrays (without considering the ECC bits in the tag arrays). The normalized areas do not include the overhead due to the remapping logic, because its relative area contribution is insignificant, as we will show in the following.

In each way of a n -way set-associative buddy cache with k divisions (see, e.g., way 0 shown in Figure 3), the only logic blocks that perhaps require some elaborations are the equality checker and the decoder. The equality checker compares the index of the way, which is a constant, and the buddy index read from the combined fault and buddy map. Although it could be further simplified because one of the two inputs is a constant, we assume a most straightforward implementation that comprises $\log n$ 2-input XNOR gates, whose outputs feed a $\log n$ -input AND gate. Assuming that a signal is available in both its true and complemented forms, the decoder requires $n - 1$ AND gates, each with $\log n$ inputs.

For simplicity, we assume a static complementary design style. Moreover, we assume that each input to a gate, be it AND, OR, NAND, NOR, or XNOR, accounts for two transistors. Although the total number of transistors obtained in this manner is just an estimate, it will be sufficiently close to the exact count in an actual implementation, which typically would have to insert additional inverters to change the polarity of a gate, or to insert additional stages of logic to restrict the number of fan-ins to a gate or the number of fan-outs of a gate.

In each division of a way, $2n + 4$ transistors are required to implement a 2-input NOR gate, n -input OR gate, and a 2-input AND gate for the computation of sel_{ij} . All divisions in a way will therefore account for $k(2n + 4)$ transistors. Moreover, in each way, the equality checker and the decoder require respectively $6 \log n$ and $2(n - 1) \log n$ transistors. The other gates in each way requires $2(k + 1) + 4(n - 1) + 14$ transistors. Therefore, the total number of transistors required in each way is $(2n + 4) \log n + 2kn + 4n + 6k + 12$. For a n -way set-associative buddy cache, the remapping logic requires a total of $n[(2n + 4) \log n + 2kn + 4n + 6k + 12]$ transistors. For $n = 16$ and $k = 4$, the remapping logic requires a total number of 5,952 transistors. For a 6T SRAM design, that is equivalent to the number of transistors required to implement 124 bytes of SRAM cells or 0.003% of 4MB L2 cache. For $n = 8$ and $k = 4$, the remapping logic requires a total number of 1,536 transistors or 32 bytes of SRAM cells, which is equivalent to 0.09% of 32KB L1 cache.

4. PERFORMANCE STUDY

The key advantage of buddying is that more functional blocks are made available. Figure 5 shows the average associativities (i.e., the average number of

Fig. 5. Average associativity of L1 and L2 buddy cache vs. perfect DDFB.

functioning blocks per set) of a L1 and a L2 buddy cache for various fault probabilities obtained through Monte Carlo simulation. The simulations were performed for fault probabilities ranging from 0.000 to 0.002. We use these published values so as to be comparable with previous works [Agarwal et al. 2005]. We assume that faults in the memory cells are predominantly caused by dopant fluctuations [Mizuno et al. 1994]. Consequently, these faults are modeled as uniformly random, occurring with a probability of p_b . For each fault probability p_b and each buddy cache configuration, we randomly inject fault (with probability p_b) into 400 instances of the buddy cache.

We used cache configurations shown in Table IV that are similar to the Intel Core 2 architecture. The L1 and L2 caches have 0 redundant sets, and four and eight redundant rows in the combined fault and buddy map, respectively. We consider four data divisions for the buddy caches. As a comparison, the average associativities of the equivalent DDFB caches with *perfect* fault maps are also plotted. The DDFB caches have 16 and 48 redundant sets for L1 and L2, respectively. The reasons for such configurations of buddy caches and DDFB caches will become apparent in Section 5. These redundant sets for DDFB caches serve two purposes. First, we use a redundant set to replace a cache set that has faults in all its cache lines. Second, we use a redundant set to improve the overall associativity; a redundant set is used to replace a cache set that has too few functional cache lines.

As shown in Figure 5, the buddy cache configurations are better able to ameliorate the negative impact of increasing fault rates. At higher-fault probabilities, the buddy cache significantly outperforms DDFB. At a bit fault probability of 0.002, DDFB can achieve only about 60% of the associativity attained by buddying.

Table IV. Architectural Configurations of Simulated SimpleScalar Processor

Conventional Cache	
Machine width	4-wide fetch, 4-wide issue, 4-wide commit
Window size	4 entry fetch queue, 128 register update unit, 8 entry load/store queue
L1 I-cache	32KB, 8-way set-associative, 64-byte block, 3 cycles hit time
L1 D-cache	32KB, 8-way set-associative, 64-byte block, 3 cycles hit time
L2 unified cache	4MB, 16-way set-associative, 64-byte block, 14 cycles hit time
Branch prediction	Combined predictor, 2,048 entry bimodal predictor, 1,024 1-bit 2-level predictor, 1,024 entry meta table
BTB	512 entry, 4-way set-associative
Memory	64-bit wide, 180 cycles first chunk, 1 cycles interchunk
TLB	16 entry (I), 32 entry (D), 4-way set-associative, 30 cycles miss latency
Function Units and Latency (total/issue)	4 Int Add(1/1), 1 Int Mult(3/1) / Div (20/19), 2 Load/Store(1/1), 4 FP Add(2/1), 1 FP Mult(4/1) / Div(12/12) / Sqrt(24/24)
Buddy Cache with 64-byte blocks	
L1 I-cache	32KB, 8-way set-associative, 64-byte block, 3 cycles hit time
L1 D-cache	32KB, 8-way set-associative, 64-byte block, 3 cycles hit time
L2 unified cache	4MB, 16-way set-associative, 64-byte block, 15 cycles hit time
DDFB Cache with 64-byte blocks	
L1 I-cache	32KB, 8-way set-associative, 64-byte block, 3 cycles hit time
L1 D-cache	32KB, 8-way set-associative, 64-byte block, 3 cycles hit time
L2 unified cache	4MB, 16-way set-associative, 64-byte block, 14 cycles hit time
DDFB Cache with 32-byte blocks and same associativities	
L1 I-cache	32KB, 8-way set-associative, 32-byte block, 3 cycles hit time
L1 D-cache	32KB, 8-way set-associative, 32-byte block, 3 cycles hit time
L2 unified cache	4MB, 16-way set-associative, 32-byte block, 14 cycles hit time
DDFB Cache with 32-byte blocks and doubled associativities	
L1 I-cache	32KB, 16-way set-associative, 32-byte block, 3 cycles hit time
L1 D-cache	32KB, 16-way set-associative, 32-byte block, 3 cycles hit time
L2 unified cache	4MB, 32-way set-associative, 32-byte block, 14 cycles hit time

One may argue that if we reduce the cache block size, we can also reduce wastage when we discard an entire cache block because of one single faulty bit in the block. Even though the choice of block size is influenced by factors such as memory latency and bandwidth [Hennessy and Patterson 2006], we nonetheless also consider two alternative DDFB cache implementations that has 32-byte cache blocks. In the first alternative DDFB cache implementation, we keep the same levels of associativity, that is, 8-way for L1 cache and 16-way for L2 cache. In order to keep the L1 and L2 caches to be of the same overall sizes as conventional L1 and L2 caches, we double the respective numbers of sets of cache blocks in both L1 and L2 caches (“2xSets”). The number of redundant cache sets for L1 and L2 caches are respectively 4 and 48. As shown in Figure 5, such a small-cache block implementation achieves associativity close to that of the buddy cache.

In the second alternative DDFB cache implementation, we doubled the associativity (“2xWays”), that is, 16-way for L1 cache and 32-way for L2 cache. The number of redundant cache sets for these L1 and L2 caches are 4 and

Fig. 6. Normalized performance at $p_b = 0.0017$.

24, respectively. Note that we did not penalize these alternative designs by increasing their access time.

To study how the average associativity correlates with performance, we performed a series of SimpleScalar simulations. We modified SimpleScalar 3.0 [SimpleScalar LLC], in particular, `sim-outorder`, to simulate “incomplete” caches. In other words, not all sets will have a full complement of cache blocks. We also used the “compressed” instruction cache of SimpleScalar, which implies that instructions are assumed to be 32-bits long. The `sim-outorder` settings used are shown in Table IV. We used the entire SPEC 2000 suite [Standard Performance Evaluation Corp.] for our experiments. In order to reduce the simulation time, we used the reduced input set for SPEC 2000 [University of Minnesota ARCTiC Labs].

In the simulations, the L1 hit latency is 3 cycles and the L2 hit latency is 14 cycles for the conventional cache and the DDFB caches (with 64-byte cache blocks and 32-byte cache blocks). In all caches, an L2 miss has a latency of at least 180 cycles. The numbers for the conventional cache were obtained from measurements by the CPU-Z tool running on an actual Intel Core 2 processor. Based on the hit latencies of the conventional cache and their delay numbers, as shown in Table III, we obtained the 15-cycle hit latency of L2 buddy cache through appropriate scaling as follows: $\lceil \frac{3339.8}{3188.6} \times 14 \rceil = 15$. From Table III, the hit latency of L1 buddy cache remains at three cycles. In other words, the buddy cache configuration has the same L1 hit latency of 3 cycles, but a L2 hit latency of 15 cycles. For the DDFB caches with 64-byte cache blocks, simulations indicated that their access times are only marginally higher than the access times of the conventional cache. Therefore, we assume that their L1 and L2 hit latencies are the same as those of the conventional cache. For simplicity, we assumed that the two alternative 32-byte DDFB configurations have the same access times as that of the conventional cache, even though simulation results suggested the possibilities of higher access times.

Figure 6 shows the performance of four fault tolerant caches at the bit fault probability of $p_b = 0.0017$ normalized to the baseline of a faultless cache. We

Fig. 7. Normalized performance at various fault probabilities.

chose to analyze this particular probability because it is close to the one reported by Agarwal et al. [2005]. The main differences are that (i) their results are for a direct mapped cache, (ii) they used a L1 of 64KB with no L2, and (iii) they used only a subset of the SPEC 2000 benchmarks, albeit running with the reference input. They reported an average performance loss of less than 5% [Agarwal et al. 2005]. This is in fairly good agreement with our results if we consider the same subset of benchmarks.

There are a number of “spikes” in Figure 6. For 179.art, while the numbers of L1 instruction and data misses were about the same, the 64-byte DDFB had more than six fold increase in L2 misses compared to the buddy configurations, pushing the L2 miss rate from 1% in the buddy caches to 6.6% in DDFB. Of the 4,096 sets in L2, the buddy cache had 185 sets with more than 10,000 misses. On the other hand, the DDFB cache had 563 such sets. These have between three to five blocks per set. On the other hand, 186.crafty suffered a 44% increase in L1 misses going from buddying to 64-byte DDFB. In all configurations, its L2 misses were insignificant. Similarly, 191.fma3d saw a 59% increase in L1 misses. In summary, for most of the benchmarks, the performance loss is marginal. The large caches, even after downsizing, were sufficient to hold most benchmarks’ working sets. However, for some benchmarks, there can be a pronounced performance impact though the reasons for the performance loss may vary. For certain benchmarks, the 32-byte DDFB configurations performs marginally better than the buddy cache. We attribute this to the temporal locality in these benchmarks. However, on average, the buddy cache outperforms all the DDFB designs. This is especially pronounced for some benchmarks like 171.swim and 173.applu that have good spatial locality.

Figure 7 shows the average normalized performance for all the benchmarks as the fault rate is increased. The higher average associativity of the buddy caches translates to a significant performance advantage, especially at higher fault probabilities. For buddy cache, the slowdown is limited to less than 8% for all the fault probabilities we studied, while for DDFB—even though the fault map is assumed to be perfect—the slowdown can be as much as 24%. At low fault

Table V. Architectural Configurations of Simulated SimpleScalar Processor with WDIS and BFIX Caches

WDIS-WDIS Cache	
L1 I-cache	16KB, 4-way set-associative, 64-byte block, 4 cycles hit time
L1 D-cache	16KB, 4-way set-associative, 64-byte block, 4 cycles hit time
L2 unified cache	2MB, 8-way set-associative, 64-byte block, 15 cycles hit time
WDIS-BFIX Cache	
L1 I-cache	24KB, 6-way set-associative, 64-byte block, 4 cycles hit time
L1 D-cache	24KB, 6-way set-associative, 64-byte block, 4 cycles hit time
L2 unified cache	3MB, 12-way set-associative, 64-byte block, 17 cycles hit time

probability, the 64-byte DDFB outperforms the 32-byte alternatives. However, after a significant number of lines are discarded, it is overtaken by the 32-byte alternatives that have higher effective associativities at the higher fault probabilities.

In Figure 7, we also showed the average normalized performance for caches designed with the WDIS and BFIX schemes. Table V shows the configurations for these caches used in the SimpleScalar simulations. The effective associativity and size of the L1 and L2 caches after applying the WDIS and BFIX schemes are also shown. As mentioned in [Wilkerson et al. 2008], the WDIS scheme and the BFIX scheme require an additional one and three clock cycles, respectively, for cache access. Due to the higher penalty, we do not consider the BFIX scheme in the L1 cache implementation, as was recommended in [Wilkerson et al. 2008]. As shown in Figure 7, the WDIS-WDIS and WDIS-BFIX combinations suffered 16.5% and 17.8% degradation in performance. Both combinations are worse than the buddy cache for all fault probabilities that we have considered for the following reasons. First, L1 caches implemented with the WDIS scheme have higher access latencies and lower associativities. Second, an L2 cache with the BFIX scheme has a much higher access latency than an L2 buddy cache. Although an L2 cache with the BFIX scheme may have a higher associativity than an L2 buddy cache at high fault probabilities, we show in the next section that it is still more desirable to use a buddy cache for the L2 cache because buddy cache has higher yield. It is also evident that both WDIS-WDIS and WDIS-BFIX combinations do not facilitate graceful degradation in performance as the number of defects increases. Both combinations also performed worse than the various DDFB caches at fault probabilities lower than 0.0015. Note that in the DDFB caches, however, we assumed that the tag array may be faulty. This may lower their associativities and degrade their performances at fault probabilities higher than 0.0015. However, for both WDIS-WDIS and WDIS-BFIX, we assumed that the tag arrays are perfect.

4.1 Energy Consumption

As shown in Figure 2, it is obvious that the combined fault and buddy map in a buddy cache introduces more leakage and access energy. Moreover, the remapping logic also consume energy during cache access. Table VI shows the energy numbers obtained from SPICE simulations of the 32KB L1 and 4MB L2 buddy caches used in SimpleScalar simulations. The setup of SPICE simulations is

Table VI. Static and Total Per-access Energy (in nJ) Obtained from SPICE Simulations of the Conventional Cache, DDFB Cache, and Buddy Cache

	32KB L1		4MB L2	
	Leakage	Per access	Leakage	Per access
Conventional	0.06748	1.05970	5.64635	2.76732
Buddy	0.06853	1.07626	5.74560	2.81859

Fig. 8. Energy overheads of buddy and perfect DDFB.

already detailed in Section 3.4. The leakage energy and total per-access energy for the L1 and L2 buddy caches are given. For comparisons, the leakage energy and total per-access energy for a conventional cache are also given. It is important to note that the buddy cache access energy does not increase significantly when compared to the conventional cache even though we may have to access data divisions from two different words. In the case of L1 cache, all banks are accessed simultaneously with the tag. Consequently, we have to consume similar access energy for the conventional cache and buddy cache. In the case of L2 cache, we assume a hierarchical word-line selection scheme [Rabaey et al. 2003]. Consequently, after a tag match, only the relevant cache block is accessed for the conventional cache. In the case of buddy cache, we have one more level of hierarchy to access at the data division level. Consequently, only the relevant divisions that form a cache block (standalone or buddied) are accessed.

Based on our SimpleScalar simulations and the energy numbers in Table VI, we computed the energy overheads of the buddy cache. The energy overheads of DDFB caches are based on the leakage and access energy of the conventional cache. This, of course, underestimates the energy overheads of DDFB caches. The results are shown in Figure 8. As one would expect, the energy overhead tracks performance. In particular, at 0.002 fault probability, the energy overhead of a buddy cache is only 37% to 66% of that of a perfect DDFB with 32-byte blocks or 64-byte blocks. Note that when the fault probability is 0, there is a marginal overhead in the DDFB cache and the buddy caches that comes from leakage energy even though the respective fault map and combined fault and buddy map are disabled.

We also computed the energy overheads of the WDIS-WDIS and WDIS-BFIX combinations and showed the results in Figure 8. The energy overheads of these two combinations are again based on the leakage energy and access energy of the conventional cache. We took into account that both WDIS and BFIX schemes are required to access two cache blocks simultaneously in the computation of energy overheads. Again, the buddy cache performs better than these two combinations. Note, however, that both WDIS and BFIX schemes are designed for low-voltage operation. In this set of experiments, we assumed that the caches are operating at the nominal supply voltage and that the defects are not due to voltage scaling but due to process parameter variations. Therefore, the energy comparison performed here may not be conclusive.

5. YIELD STUDY

In this section, we will study a limiting case of performance degradation, namely the catastrophic failure of the chip due to the failure of the cache. This is closely related to the “yield” of the chip. Unfortunately, we do not have knowledge about how yield is defined by the chip vendors. Rather, without loss of generality, we will use the following definition of yield: A set in a conventional or buddy cache is functional if it has at least one functional way. A cache with m sets is functional if after using part or all of its redundant resources, it has m functional sets. The yield of a cache is the probability that it is functional.

First, we shall present a detailed analytical yield model for the buddy cache. We use the notation introduced in Section 3.5: An n -way set-associative cache with m sets, with tag size t and k data divisions, each of which has d data bits.

The probability that a cache block (both tag and data) is faulty is $P_{\text{block-faulty}} = (1 - (1 - p_b)^{t+dk})$. For two cache blocks to be buddy blocks, at least one of the two corresponding tags must be operational. Moreover, the data divisions at the same position in the two buddy blocks cannot be both faulty. Therefore, the probability that the pairing of two cache data blocks in a set produces an operational block is $P_{\text{buddy-block}} = (1 - (1 - (1 - p_b)^t)^2)(1 - (1 - (1 - p_b)^d)^2)^k$. Within a set, there are altogether $\binom{n}{2}$ possible pairs of cache blocks. As defined, a set is nonfunctional only if all blocks are faulty and none of these pairings can produce an operational buddy cache block. Hence, the probability that a set is functional is $P_{\text{set}} = 1 - P_{\text{block-faulty}}^n (1 - P_{\text{buddy-block}})^{\frac{n(n-1)}{2}}$. For a cache with R redundant sets, at most R sets can be faulty for the cache to be operational. The probability that the buddy cache is operational, with fault map and buddy map being perfect, is therefore

$$P_{\text{buddy-cache}} = \sum_{i=0}^R \binom{m+R}{i} P_{\text{set}}^{m+R-i} (1 - P_{\text{set}})^i.$$

For a buddy cache, a set is functional if the following conditions hold: (i) there must be at least one operational cache block or one operational pair of buddy cache blocks; and (ii) the corresponding entries in the combined fault and buddy map must be operational. Condition (ii) implies that there cannot be a cache block whose tag fault bit and buddy index both have faults or if the buddy index is faultless, the fault bits are all faulty. The probability that fault bits and buddy

Fig. 9. Validation of analytical yield model with Monte Carlo simulations.

index of a particular cache block would render the corresponding row in the combined fault and buddy map unusable is $P_{\text{FB-block-faulty}} = (1 - (1 - p_b)^{k+1})(1 - (1 - p_b)^{\log n}) + p_b^{k+1}(1 - p_b)^{\log n}$. For better performance, the combined fault and buddy map should be arranged in an aspect ratio that is closer to being a square. Let s denote the number of sets of cache blocks guarded in a row of a “squarish” combined fault and buddy map. The probability that a row in the combined fault and buddy map is usable is, therefore, $P_{\text{FB-row-usable}} = (1 - P_{\text{FB-block-faulty}})^{ns}$. Assuming that we have r redundant rows in the combined fault and buddy map, the probability that the combined fault and buddy map are usable is

$$P_{\text{FB-maps-usable}} = \sum_{i=0}^r \binom{\frac{m}{s} + r}{i} P_{\text{FB-row-usable}}^{m/s+r-i} (1 - P_{\text{FB-row-usable}})^i.$$

Hence, for a given p_b , we can compute an upper bound for the yield of the buddy cache as follows:

$$P_{\text{buddy-cache-FB}}(p_b) \leq P_{\text{FB-maps-usable}} P_{\text{buddy-cache}}. \quad (4)$$

Using the same Monte Carlo technique described earlier, we studied the yield of the buddy cache. Here, we consider the addition of 0, 1, 2, or 4 redundant rows to the buddy cache. It is evident from Figure 9 that the analytical model tracks the results from Monte Carlo simulation of 4MB L2 buddy cache with 4 data divisions very well, thereby cross validating each other.

The results in Figure 9 suggest that in order to achieve a high yield at a high-fault probability of 0.002, it is necessary to include at least four redundant rows in the combined fault and buddy map of a buddy cache. In fact, we shall assume a buddy cache configuration with 8 redundant rows to have a almost perfect yield (or $\geq 99.99\%$). The numbers of redundant rows/sets in L1 and L2 buddy caches used in Section 4 are determined based on this observation.

Fig. 10. Yield of a DDFB cache with various levels of redundancy.

The yield of a DDFB cache with 64-byte cache blocks at various fault probabilities is plotted in Figure 10. Similar to that of a buddy cache, the yield plot is obtained from an analytical formula for the yield of DDFB cache, which is adapted from an analytical yield model published in Agarwal et al. [2005]. As shown in Figure 10, the overall yield of DDFB cache is very low. Worse, the yield of DDFB cache with a large number of redundant rows added to the fault map does not improve significantly. In contrast, the advantage of buddy cache over DDFB cache is substantial. Given this, one may argue for a more robust implementation for the fault maps of DDFB caches. Therefore, we use a triple-modular redundancy of the DDFB fault map. In other words, three copies of the fault map are maintained, and a majority voting scheme is used to decide on the correct output. Both analytical yield model and Monte Carlo simulations indicated that with at least 8 redundant rows in the fault map (with triple-modular redundancy), the fault map of a DDFB cache can be considered to be “perfect” (or $\geq 99.99\%$). However, if we do not add redundant cache sets, the yield performance of DDFB cache with a perfect fault map is much worse than that of a buddy cache without redundant cache sets. Again, both analytical model and Monte Carlo simulations suggested that with at least 8 redundant cache sets, we can achieve almost 100% yield for DDFB cache. Therefore, in the performance study of DDFB cache with a perfect fault map in Section 4, we considered a DDFB L2 cache with 16 redundant rows in the fault map (with triple-modular redundancy) and 48 redundant sets in the cache. Note that we use 48 redundant sets in the DDFB cache instead of just 16 so that both L2 DDFB cache and L2 buddy cache have the similar numbers of SRAM cell. In other words, they have the same area overhead. With 16 redundant fault map rows and 48 redundant cache sets, we observed a similar yield performance for a 16-way DDFB cache with 32-byte cache blocks. For a 32-way DDFB cache with 32-byte cache blocks, we consider 16 redundant fault map rows and

Fig. 11. Yield of caches implemented with the WDIS scheme and the BFIX scheme.

24 redundant cache sets. In the case of L1 caches, all three versions of DDFB caches (with triple modular redundancy of fault maps) have higher numbers of SRAM cells than buddy cache. Even though all these DDFB caches have similarly very good yield performances when compared with the buddy cache, we have already shown in Section 4 that a buddy cache has superior performance when compared to a DDFB cache.

Figure 11 shows the yield of 4M L2 16-way set-associative caches implemented with the WDIS scheme and the BFIX scheme. (The effective size and associativity are 2M and 8-way, respectively, for the WDIS scheme and 3M and 12-way, respectively, for the BFIX scheme.) The yield levels were obtained with both analytical formulas and Monte Carlo simulations, and it is evident that they cross-validate each other. The plot for a cache implemented with the WDIS scheme is obtained with no redundant cache sets, because the fault map overhead of such a scheme is already more expensive than that of the buddy cache. Although the yield is perfect for the WDIS scheme, it is important to keep in mind that we assume that the fault map is perfect, as was assumed by Wilkerson et al. [2008].

The BFIX scheme does not require a fault map. Therefore, we provide 72 redundant cache sets to the cache implemented with the BFIX scheme, in order to improve the yield. As shown in Figure 11, the yield drops quickly to 0 when the fault probability increases beyond 0.001. Therefore, even though the BFIX scheme in theory may have an effective associativity of 12 for a 16-way

Fig. 12. Injection of correlated faults.

set-associative cache, we may not have a functional cache in practice. We believe that the yield of the BFIX scheme can be improved with triple modular redundancy of the patches. However, that is beyond the scope of this work.

5.1 Spatially Correlated Faults

Thus far, we have assumed that faults are uniformly random and that they occur with a probability p_b . Now, we consider a more general scenario when faults may occur in a cluster due to spatial correlation. In [Ozdemir et al. 2006], it was argued that due to spatial correlation, other cells within the same way of a faulty cell are also likely to be faulty; it may therefore be better to shut down an entire way. As it is difficult to derive analytical yield expression, we relied on Monte Carlo simulations to investigate how spatially correlated faults may affect the yield of all the cache schemes that we have considered thus far.

In each Monte Carlo simulation, we first inject faults into the combined fault and buddy map, tag array, and data array of a buddy cache with probability p_b . For each injected fault, we also turn SRAM cells next to it faulty according to one of the nine fault configurations, as shown in Figure 12. We label each fault configuration by “ i - j ” for some combinations of $i, j \in \{0, 1\}$, where i is the number of additional faults to the left (or right) of the injected fault and j is the number of additional faults above (or below) the inject fault. The fault configuration “0-0” for example, does not insert additional faults, whereas the fault configuration “1-1” adds 4 faulty SRAM cells in close proximity to each injected faults. Let N be the total number of faults in a buddy cache under the fault configuration “0-0” with a fault probability p_b . With the same fault probability p_b , the number of faults in a similar cache under the fault configuration “1-1” would be around $5N$. In order to have the same overall fault rate in both fault configurations, we therefore used a properly scaled fault probability ($p_b/5$, for example) for the fault configuration “1-1” for every fault probability p_b used in the Monte Carlo simulations of caches with fault configuration “0-0.”

In Figure 13(a) we plot the yield of 4MB L2 buddy cache with 8 redundant rows in the combined fault and buddy map under fault configurations; we already showed for fault configuration “0-0.” the average associativity in Section 4 and the yield in the earlier part of this section. For comparison, we also plot the yield of 4MB L2 DDFB cache with 64-byte blocks. Here, we assume that the DDFB cache has 16 redundant rows in the fault map and 48 redundant cache sets. For each fault configuration, we can observe that buddy cache and DDFB cache have similar yield at low fault probabilities. The yield of buddy cache decreases slightly at high fault probabilities, but the degradation of the yield of DDFB cache is quite pronounced. The yield of buddy cache can be improved to about 97% and 99% by adding 2 and 4 redundant rows to the combined

(a)

(b)

Fig. 13. Comparisons of the (a) yield and (b) associativity of 4MB L2 buddy cache with 4MB L2 DDFB cache with 64-byte blocks under various correlated fault configurations.

fault and buddy map, respectively. Moreover, the buddy cache has significantly higher average associativities than its DDFB counterpart under all fault probabilities that we investigated, as shown in Figure 13(b). This translates into higher performance as shown in Section 4.

We also compare the yield and associativity of the buddy cache with 16-way DDFB cache and 32-way DDFB cache, both of which have 32-byte cache blocks in Figure 14 and Figure 15, respectively. As shown in Figure 14(a) and

(a)

(b)

Fig. 14. Comparisons of the (a) yield and (b) associativity of 4MB L2 buddy cache with 4MB, 16-way L2 DDFB cache with 32B-byte blocks under various correlated fault configurations.

Figure 15(a), the buddy cache has similar yield as these DDFB caches at most fault probabilities. As pointed out earlier, the yield of the buddy cache drops slightly when the fault probabilities are high. When we increase the number of redundant rows in the combined fault and the buddy map to 10, the yield of the buddy cache is as good as these DDFB caches. With 12 redundant rows in the combined fault and buddy map, the yield of the buddy cache outperforms these DDFB caches. The levels of associativity of the 16-way DDFB cache and buddy cache are similar. Based on the results in Section 4, we can conclude that buddy cache would have better performance than 16-way DDFB cache with

(a)

(b)

Fig. 15. Comparisons of the (a) yield and (b) associativity of 4MB L2 buddy cache with 4MB, 16-way L2 DDFB cache with 32B-byte blocks under various correlated fault configurations.

32-byte cache blocks. The levels of associativity of 32-way DDFB cache are about twice those of buddy cache. Again, we can conclude that the buddy cache would outperform a 32-way DDFB cache with 32-byte cache blocks.

Finally, we compare the yield of the buddy cache with caches implemented with the WDIS and BFIX schemes under various fault configurations (see Figure 16). It is obvious that the buddy cache is more robust than both schemes.

6. VARIANTS FOR TRADING OFF AREA AND PERFORMANCE

For an n -way buddy cache with m sets of cache blocks, there is an area overhead of $mn(k + 1 + \log n)$ bits for the buddy cache. While it is clear how the overhead

Fig. 16. Comparisons of the yield of 4MB L2 buddy cache with 4MB, 16-way L2 cache implemented with (a) the WDIS scheme and (b) the BFIX scheme under various correlated fault configurations.

Fig. 17. (a) Normalized total area of the buddy cache and (b) expected yield for various options.

in the fault map can be reduced by decreasing the number of data divisions, we now present a technique to reduce the area requirement of the buddy map. In the previous discussion, we have assumed that a faulty cache block can find a buddy within the entire set, and thus, it requires $\log n$ bits of buddy index to record its buddy. We can reduce the overhead due to the buddy map by restricting the neighborhood in which a faulty cache block can find its buddy.

Consider for example, an n -way buddy cache, we can partition the n cache blocks within a set into two buddying groups (of $n/2$ cache blocks each) according to the cache block indices: Cache block B_i resides in group $\lfloor 2i/n \rfloor$. If cache block B_i is faulty, it can form a buddy pair with another faulty cache block B_j only if $\lfloor 2i/n \rfloor = \lfloor 2j/n \rfloor$, that is, the two compatible faulty cache blocks reside in the same buddying group. In this example, the overhead of buddy map can be reduced by a factor of $1/\log n$. To show the area reduction due to number of data divisions and restriction in buddying options, Figure 17(a) plots the total area of buddy cache (inclusive of data, tag, fault map, and buddy map) for different combinations, normalized by the area of a conventional 4MB 16-way set-associative cache. Among all the combinations that we have considered for this study, the combined fault and buddy map occupies between 0.6% to 2.6% of the area required of a conventional L2 cache. In both plots, we did not consider logic, as its relative area contribution is insignificant.

Figure 17(b) plots the expected yield of buddy cache for various combinations of data divisions and buddying options. Agarwal et al. [2005] performed Monte Carlo simulation of 1000 chips (64KB cache) in 0.9V to obtain fault statistics for 45nm PTM technology. The nominal threshold voltage and the standard deviation of the intradie variation were assumed to be 0.22V and 33mV, respectively. Based on the chip counts in Figure 4 of Agarwal et al. [2005] and the probability that a chip is functional (Eq. 4) for a given fault probability p_b , the expected yield of the buddy cache, given a certain process technology is

$$E(\text{yield})_{\text{buddy-cache}} = \frac{1}{N_{\text{total}}} \sum P_{\text{buddy-cache-FB}}(p_b) N_{\text{chips}}(p_b),$$

Fig. 18. (a) Average associativity and (b) normalized associativity of L2 buddy cache, for various combinations of data divisions and buddying options.

where N_{total} is the total number of chips made, and $N_{\text{chips}}(p_b)$ is the number of chips with fault probability p_b (or equivalently, with $p_b \times 64\text{KB}$ faulty cells). It is evident from the plot that the buddy cache incurs a low-area overhead for achieving 99.99% expected yield consistently.

The average associativity of the buddy cache with various combinations of data divisions and buddying options are shown in Figure 18(a). While increasing the number of data divisions and the buddying group size improves the average associativity, it is at the expense of area. Here, we propose a metric to evaluate the efficacy of a data division and buddying combination:

$$\text{Normalized Associativity} = \frac{\text{Ave Associativity} \times \text{E}(\text{yield})_{\text{buddy-cache}}}{\text{Normalized Area}},$$

where “Ave Associativity” can be obtained from Figure 18(a), “ $\text{E}(\text{yield})_{\text{buddy-cache}}$ ” from Figure 17(b), and “Normalized Area” from Figure 17(a). Figure 18(b) plots the normalized associativities for various data division and buddying combinations. It should also be evident that in terms of the number of functional blocks in a cache, the buddy cache outperforms various versions of DDFB cache and caches implemented with the WDIS and BFIX schemes.

7. CONCLUSION

In this article, we addressed the problems of process variations degradations on large, set-associative caches by proposing the buddy cache. Compared to previous DDFB cache proposals, the buddy cache is able to reuse faulty cache blocks by buddying two such blocks that do not have faults in the same bit position. This reuse results in the increase of the number of usable cache blocks thereby making the processor as a whole more resilient to performance degradation. At a bit fault probability of 0.002, a processor with buddy caches is less than 8% slower than a baseline with faultless caches over the large suite of benchmarks we used. On the other hand, in a processor with DDFB caches,

we saw a 24% degradation in performance. For a cache implemented with the WDIS-WDIS combination or the WDIS-BFIX combination, the degradations in performance are 16.5% and 17.8%, respectively. The performance gap is even more significant at lower fault probabilities.

Furthermore, the buddy cache's remapping logic can tolerate faults much better. As a result, yield is significantly improved. Again, at the same 0.002-bit fault probability, the buddy cache's yield is near 99% while the DDFB's yield (without triple modular redundancy of fault map) is at a miserable 1.4%—a $70\times$ improvement.

With circuit level simulations, we established that buddying has a negligible effect on the critical path of the L1 cache, and adds at most one cycle of penalty to L2 access. In this article, therefore, we have dealt with all aspects—design, implementation, timing, area, performance, energy, and yield—of the buddy cache.

The buddy cache goes beyond handling errors caused by defects in silicon, lithographic failures, and process variation at the time of manufacture. The fault and buddy maps of the buddy cache are reinitialized each time the processor reboots. This allows the processor to also deal with the accumulation of faults caused by device ageing. Like the recently proposed WDIS and BFIX schemes, it can also be used for lowering supply voltages. Compared to WDIS, BFIX and the state-of-the-art DDFB schemes, we have shown that the performance degradation is less pronounced and more gradual. At the same time, any ECC mechanism present in the cache is free to deal with transient errors. We believe that the buddy cache can contribute toward fault resilient processors designed in the nanometer regime.

REFERENCES

- AGARWAL, A., PAUL, B. C., MAHMOODI, H., DATTA, A., AND ROY, K. 2005. A process-tolerant cache architecture for improved yield in nanoscale technologies. *IEEE Trans. VLSI Syst.* 13, 1, 27–38.
- ARIMILLI, R. K., DODSON, J. S., LEWIS, J. D., AND SKERGAN, T. M. 1999. Cache array defect functional bypassing using repair mask. U.S. Patent Number 5,958,068.
- BORKAR, S. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE MICRO* 25, 6, 10–16.
- BORKAR, S., KARNIK, T., NARENDRA, S., TSCHANZ, J., KESHAVARZI, A., AND DE, V. 2003. Parameter variations and impact on circuits and micro-architecture. In *Proceedings of the 40th Design Automation Conference (DAC'03)*. 338–342.
- CAO, Y., SATO, T., SYLVESTER, D., ORSHANSKY, M., AND HU, C. 2000. New paradigm of predictive MOSFET and interconnect modeling for early circuit design. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. 201–204.
- CHOI, S. H., PAUL, B. C., AND ROY, K. 2004. Novel sizing algorithm for yield improvement under process variation in nanometer technology. In *Proceedings of the 41st Design Automation Conference (DAC'04)*. 454–459.
- DATTA, A., BHUNIA, S., MUKHOPADHYAY, S., BANERJEE, N., AND ROY, K. 2005. Statistical modeling of pipeline delay and design of pipeline under process variation to enhance yield in sub-100nm technologies. In *Proceedings of Design, Automation and Test in Europe (DATE'05)*. 926–931.
- FISCHER, T., OLBRICH, A., GEORGAKOS, G., LEMAITRE, B., AND SCHMITT-LADSIEDEL, D. 2007. Impact of process variations and long term degradation on 6T-SRAM cells. In *Advances in Radio Science* 5, 321–325.

- FUJIMOTO, Y. 2000. Cache memory having flags for inhibiting rewrite of replacement algorithm area corresponding to fault cell and information processing system having such a cache memory. U.S. Patent Number 6,145,055.
- HENNESSY, J. L. AND PATTERSON, D. A. 2006. *Computer Architecture: A Quantitative Approach* 4th Ed. Morgan-Kaufmann, San Francisco, CA.
- KURDAHI, F. J., ELTAWIL, A. M., PARK, Y.-H., KANJ, R. N., AND NASSIF, S. R. 2006. System-level SRAM yield enhancement. In *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)*. 179–184.
- LEE, H., CHO, S., AND CHILDERS, B. R. 2007. Performance of graceful degradation for cache faults. In *Proceedings of the IEEE Symposium on VLSI (ISVLSI'07)*. 409–415.
- LIANG, X. AND BROOKS, D. 2006. Microarchitecture parameter selection to optimize system performance under process variation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'06)*. 429–436.
- MARCULESCU, D. AND TALPES, E. 2005. Variability and energy awareness: a micro-architecture-level perspective. In *Proceedings of the 42nd Design Automation Conference (DAC'05)*. 11–16.
- MCCLURE, D. 1997. Method and system for bypassing a faulty line of data or its associated tag of a set associative cache memory. U.S. Patent Number 5,666,482.
- MIZUNO, T., OKUMURA, J., AND TORIUMI, A. 1994. Experimental study of threshold voltage fluctuation due to statistical variation of channel dopant number in MOSFET's. *IEEE Trans. Electron Devices* 41, 11, 2216–2221.
- MUKHOPADHYAY, S., MAHMOODI, H., AND ROY, K. 2004. Statistical design and optimization of SRAM cell for yield enhancement. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'04)*. 10–13.
- OZDEMIR, S., SINHA, D., MEMIK, G., ADAMS, J., AND ZHOU, H. 2006. Yield-aware cache architectures. In *Proceedings of the 39th International Symposium on Micro-Architecture (MICRO-39)*. 15–25.
- Pellston 2006. Intel's newest Quad Xeon MP versus HP's DL585 Quad Opteron. <http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=2872&p=2>.
- POWELL, M. D., AGARWAL, A., VIJAYKUMAR, T. N., FALSAFI, B., AND ROY, K. 2001. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th International Symposium on Micro-Architecture (MICRO-34)*. 54–65.
- RABAAY, J. M., CHANDRAKASAN, A., AND NIKOLIĆ, B. 2003. *Digital Integrated Circuits: A Design Perspective* 2nd Ed. Prentice-Hall, Upper Saddle River, NJ.
- SHIRVANI, P. P. AND MCCLUSKEY, E. J. 1999. PADdded cache: A new fault-tolerance technique for cache memories. In *Proceedings of the 17th IEEE VLSI Test Symposium (VTS'99)* IEEE Computer Society, USA, 440.
- SIMPLESCALAR LLC. The SimpleScalar toolset. <http://www.simplescalar.com>.
- STANDARD PERFORMANCE EVALUATION CORP. SPEC CPU2000 V1.3. <http://www.spec.org/cpu2000>.
- TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. P. 2006. CACTI 4.0. Tech. rep., HP Western Research Labs.
- TSCHANZ, J., BOWMAN, K., AND DE, V. 2005. Variation-tolerant circuits: Circuit solutions and techniques. In *Proceedings of the 42nd Design Automation Conference (DAC'05)*. 762–763.
- UNIVERSITY OF MINNESOTA ARCTIC LABS. Minnespec: A new SPEC benchmark workload for simulation-based computer architecture research. <http://www.arctic.umn.edu/minnespec/index.shtml>.
- WILKERSON, C., GAO, H., ALAMELDEEN, A. R., CHISHTI, Z., KHELLAH, M., AND LU, S.-L. 2008. Trading off cache capacity for reliability to enable low voltage operation. *SIGARCH Comput. Archit. News* 36, 3, 203–214.
- XIONG, J., TAM, K., AND HE, L. 2005. Buffer insertion considering process variation. In *Proceedings of Design, Automation and Test in Europe (DATE'05)*. 970–975.
- ZHANG, C., VAHID, F., YANG, J., AND NAJJAR, W. 2005. A way-halting cache for low-energy high-performance systems. *ACM Trans. Archit. Code Optim.* 2, 1, 34–54.

Received May 2008; revised November 2008; accepted February 2008